Linked List
Data Structures & Algorithms
Syed Mudassar Alam

Linked Lists

Outline

- Introduction
- Insertion Description
- Deletion Description
- Basic Node Implementation
- Conclusion

Outline

- Introduction
- Insertion Description
- Deletion Description
- Basic Node Implementation
- Conclusion

Introduction

- Definitions
  - Lists and arrays
  - Nodes and pointers
  - Single Linked Lists
  - Double Linked Lists
  - Circular Lists
- Advantages

Defination:

A **linked list** is a series of *connected* ***nodes*** (or **links**) where each node is a *data structure*.

Dynamically allocated data structures can be **linked together** to form a **chain**.

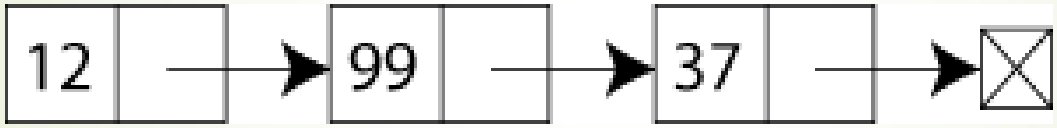
A linked list can grow or shrink in size as the program runs.This is possible because the nodes in a linked list are dynamically allocated.

Introduction

- Definitions
  - Lists and arrays
  - Nodes and pointers
  - Single Linked Lists
  - Double Linked Lists
  - Circular Lists
- Advantages

Lists and arrays

- Lists:



Lists and arrays

- Arrays:

{Size of the following array is = 4}

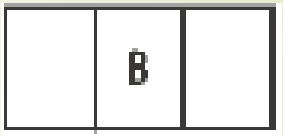
Index	0	1	2	3
Value	44	5	96	3

Introduction

- Definitions
  - Lists and arrays
  - Nodes and pointers
  - Single Linked Lists
  - Double Linked Lists
  - Circular Lists
- Advantages

Nodes and pointers

- Each nodes contains the Data Field and a reference field (Pointers) that points to next node.

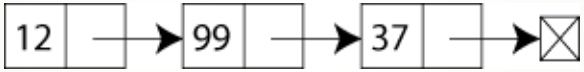


- The last nodes contains the null value in its second part because there is no node after it.

Introduction

- Definitions
  - Lists and arrays
  - Nodes and pointers
  - Single Linked Lists
  - Double Linked Lists
  - Circular Lists
- Advantages

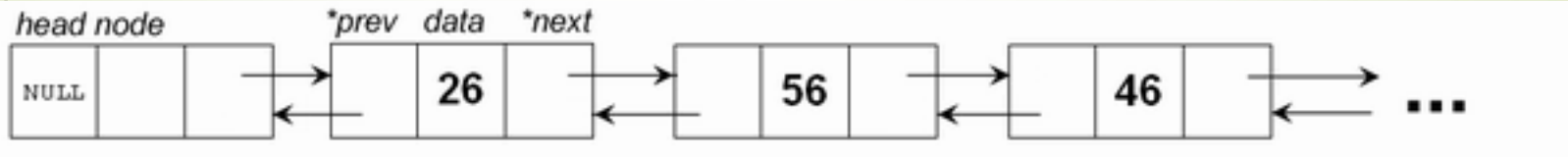
Single linked lists



Introduction

- Definitions
  - Lists and arrays
  - Nodes and pointers
  - Single Linked Lists
  - Double Linked Lists
  - Circular Lists
- Advantages

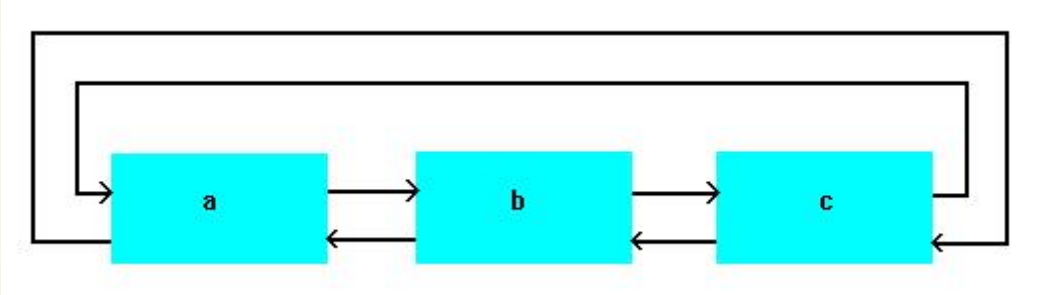
Double Linked Lists



Introduction

- Definitions
  - Lists and arrays
  - Nodes and pointers
  - Single Linked Lists
  - Double Linked Lists
  - Circular Lists
- Advantages

Circular Lists



Introduction

- Definitions
  - Lists and arrays
  - Nodes and pointers
  - Single Linked Lists
  - Double Linked Lists
  - Circular Lists
- Advantages

Advantages

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.

a) A linked list can easily grow and shrink in size -

- The programmer doesn't need to know how many nodes will be in the list. They are created in memory as needed.

b) Speed of insertion or deletion from the list -

- Inserting and deleting elements into and out of arrays requires moving elements of the array. When a node is inserted, or deleted from a linked list, none of the other nodes have to be moved

Basic Operations of a Linked List

- As an abstract data type,
  A **list** is a finite sequence (possibly empty) of elements with basic operations that vary from one application to another.

  Basic operations commonly include:
- **Construction:** Allocate and initialize a list object (usually empty)
- **Empty:** Check if list is empty

Basic Operations of a Linked List

- **Traverse:** Visit each element of the list in order stored
- **Insert:** Add an item to the list (at any point)
- **Delete:** Remove an item from the list (at any point)

Basic Operations of a Linked List

- **Construction:** first = *null_value*;
- **Empty:** return (first == *null_value*)
- **Traverse:**

```
ptr = first;
while (ptr != null_value)

{ Process data part of node pointed to by ptr;
ptr = address of next element
 // contained in next part of
 // node pointed to by ptr;
 }
```

Basic Operations of a Linked List

- **Insert & Delete:** Both operations necessitate the changing of links.

  The address of the predecessor of the node to be added or removed needs to be known.

Outline

- Introduction
- Insertion Description
- Deletion Description
- Basic Node Implementation
- Conclusion

Insertion Description

- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion Description

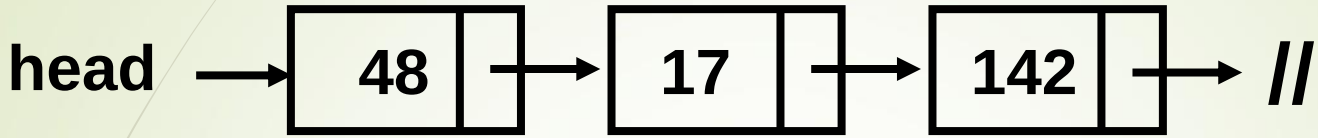
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle (any) of the list

Insertion at the top

Steps:

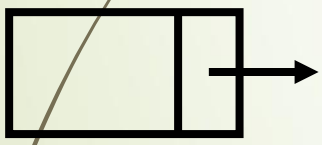
- Create a Node
- Set the node data Values
- Connect the pointers

Insertion Description

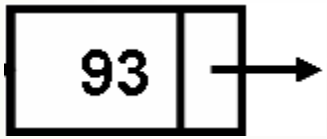


- Follow the previous steps and we get

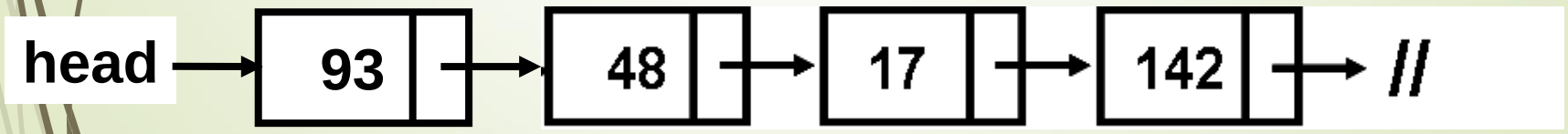
Step 1



Step 2



Step 3



Insertion Description

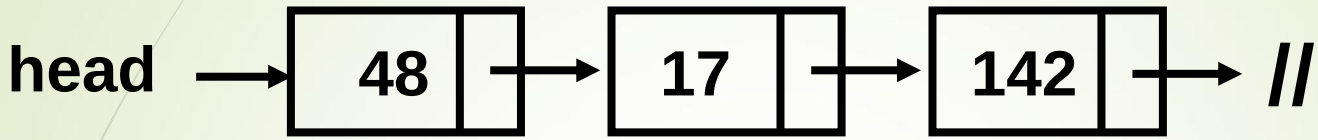
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion at the end

Steps:

- Create a Node
- Set the node data Values
- Connect the pointers

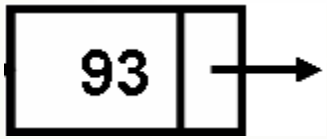
Insertion Description



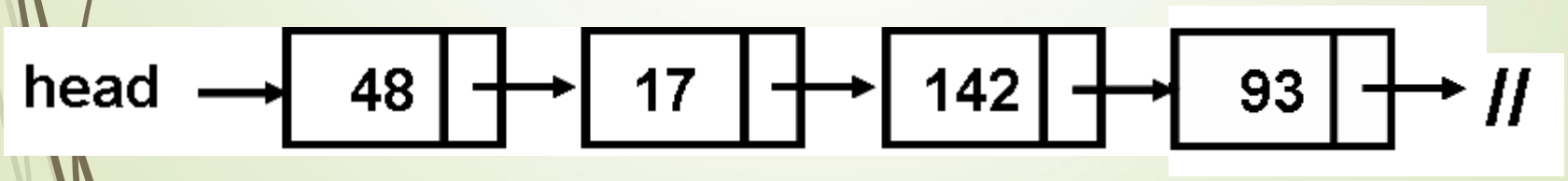
- Follow the previous steps and we get

Step 1

Step 2



Step 3



Insertion Description

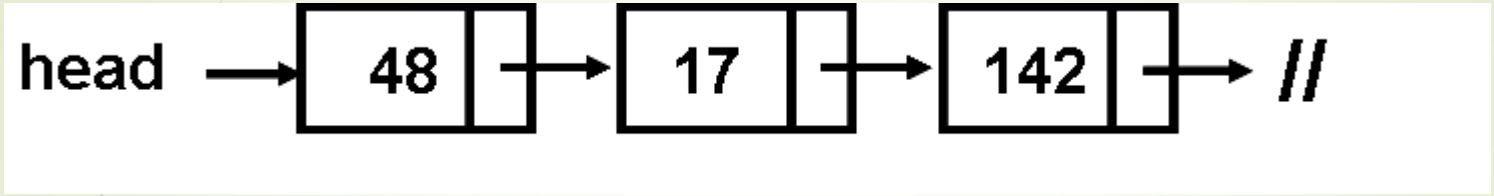
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion in the middle

Steps:

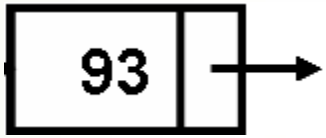
- Create a Node
- Set the node data Values
- Break pointer connection
- Re-connect the pointers

Insertion Description

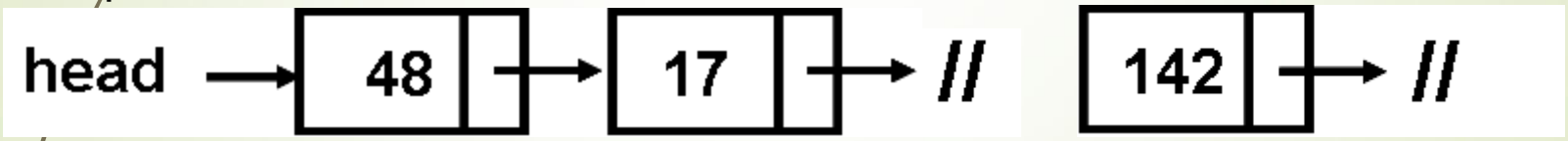


Step 1

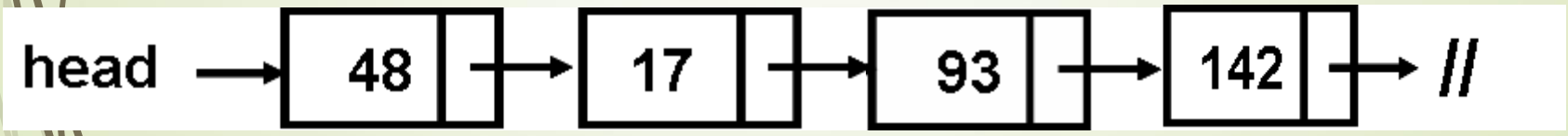
Step 2



Step 3



Step 4



Outline

- Introduction
- Insertion Description
- Deletion Description
- Basic Node Implementation
- Conclusion

Deletion Description

- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deletion Description

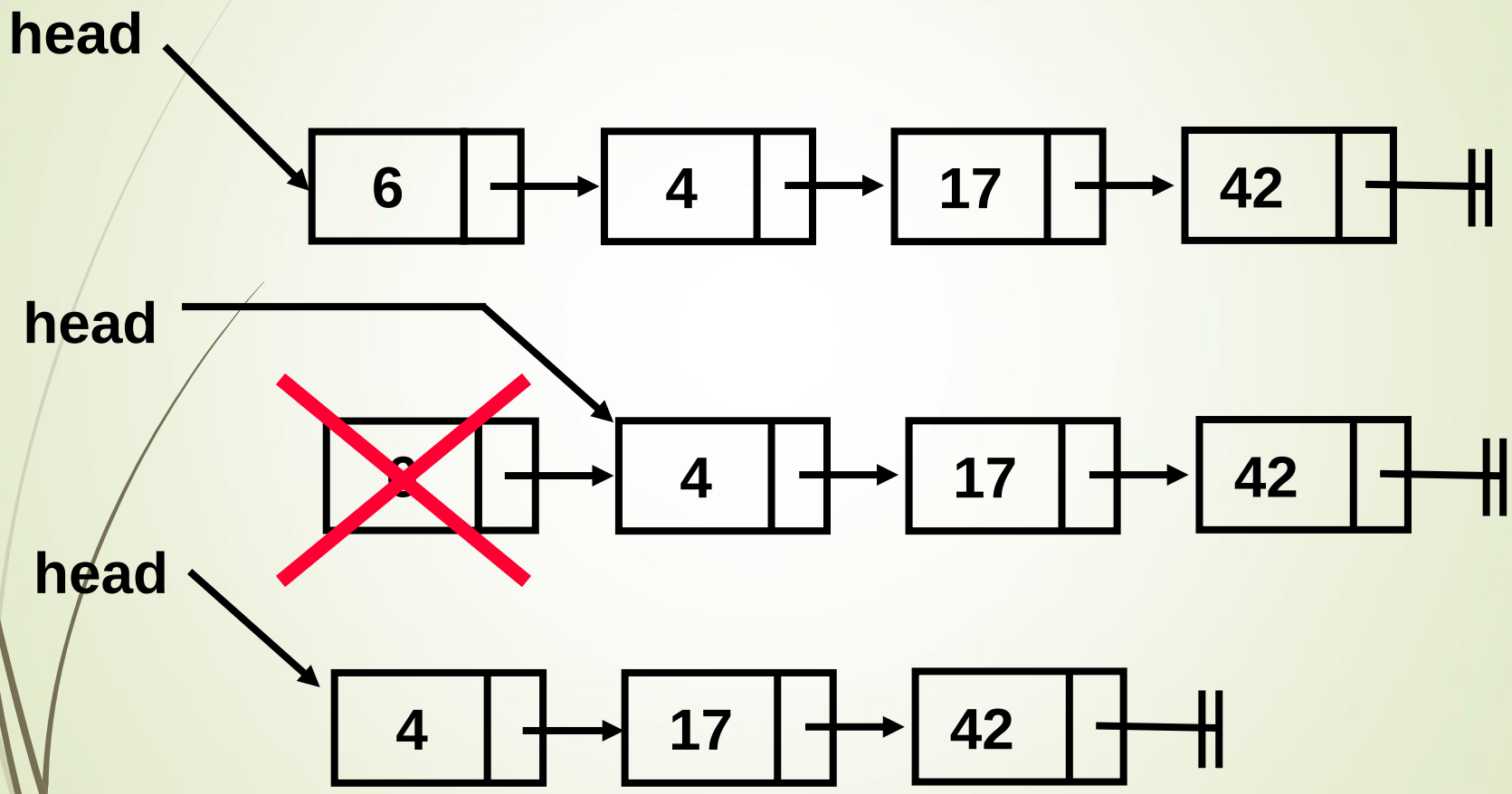
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the top

Steps

- Break the pointer connection
- Re-connect the nodes
- Delete the node

Deletion Description



Deletion Description

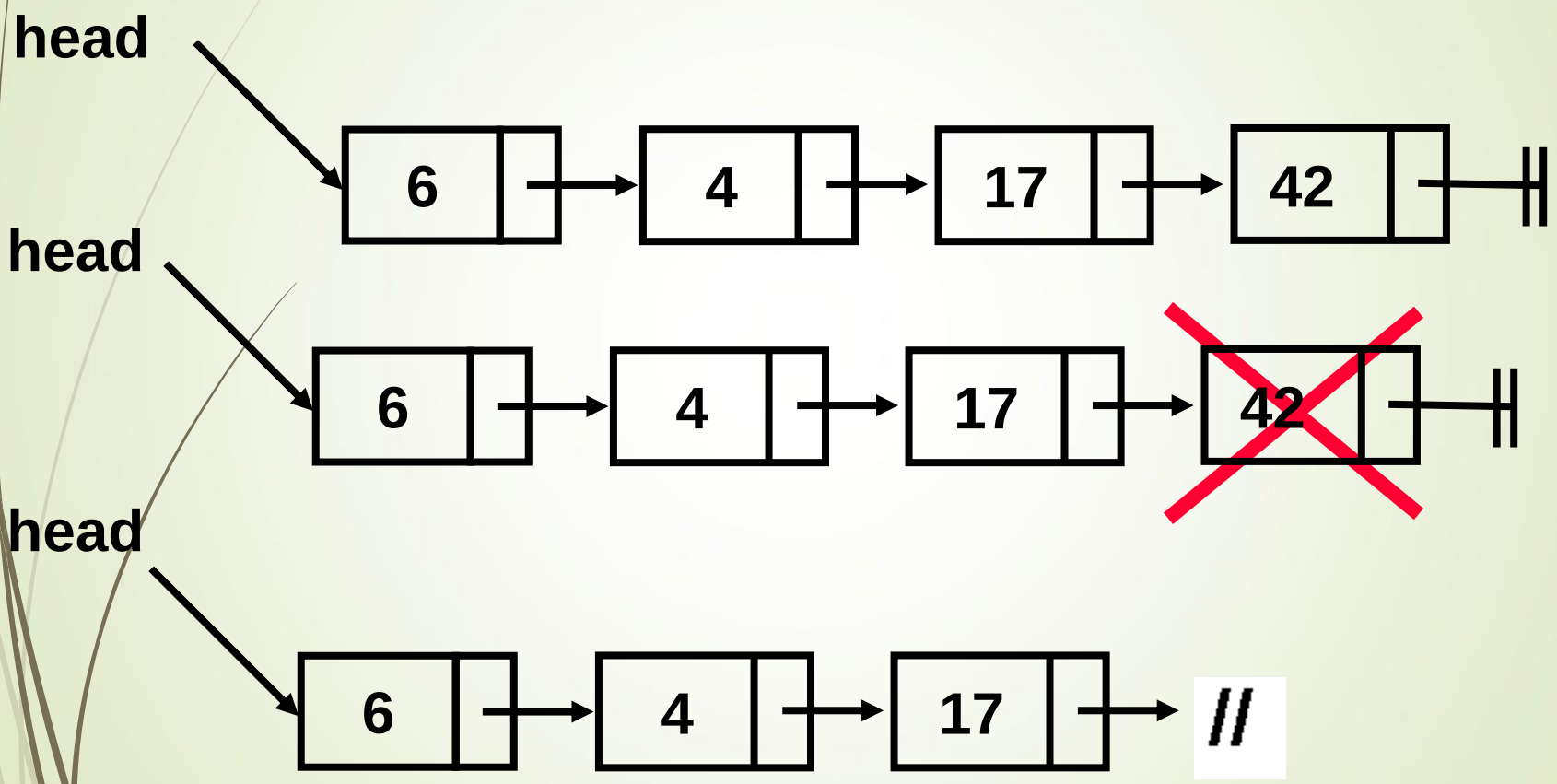
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the end

Steps

- Break the pointer connection
- Set previous node pointer to NULL
- Delete the node

Deletion Description



Deletion Description

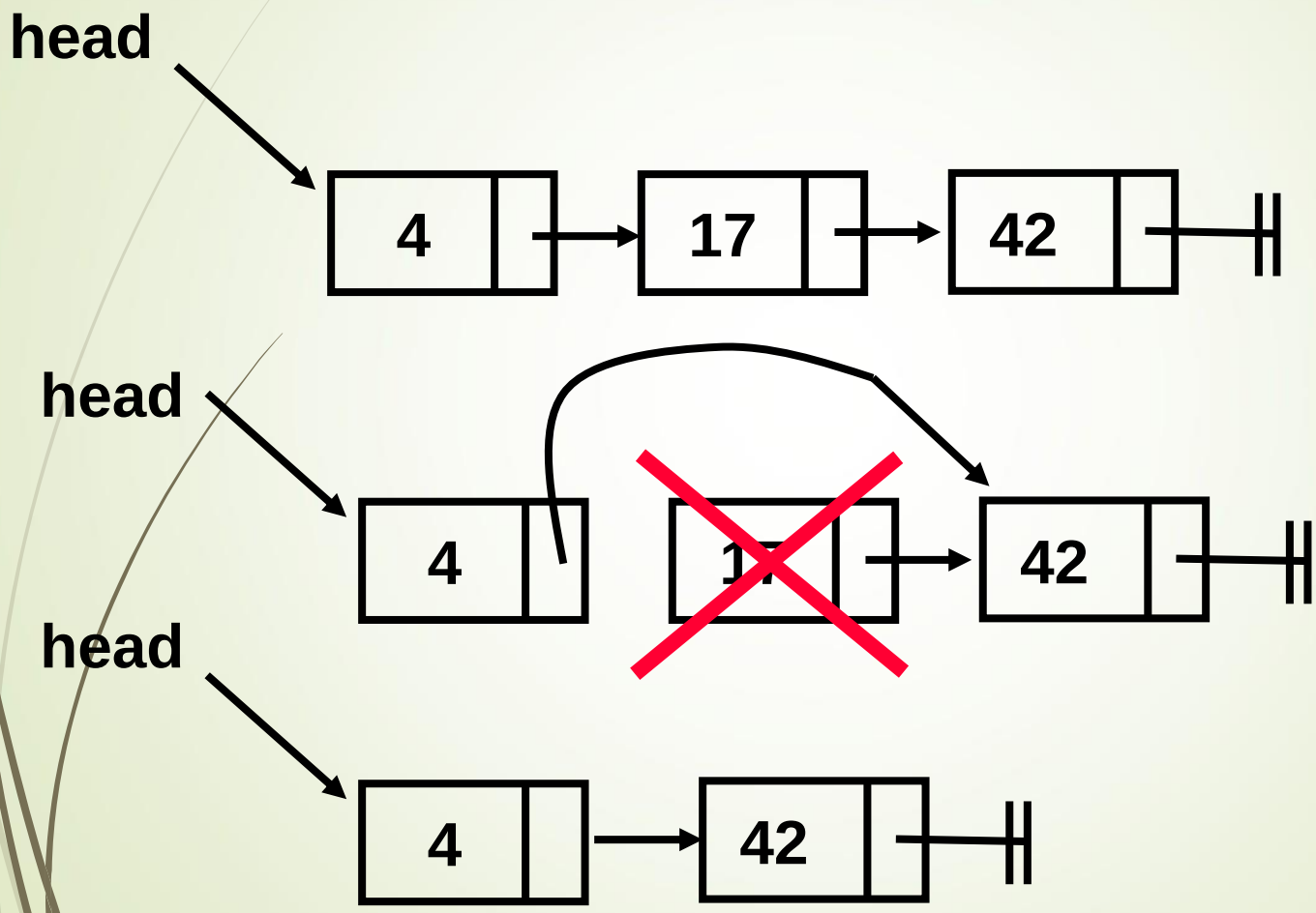
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the Middle

Steps

- Set previous Node pointer to next node
- Break Node pointer connection
- Delete the node

Deletion Description



Outline

- Introduction
- Insertion Description
- Deletion Description
- Basic Node Implementation
- Conclusion

Basic Node Implementation

The following code is written in C++:

```
Struct Node
{
    int data;    //any type of data could be another struct
    Node *next;   //this is an important piece of code
                  "pointer"
};
```

Outline

- Introduction
- Insertion Description
- Deletion Description
- Basic Node Implementation
- Conclusion

Adding a Node

There are four steps to add a node to a linked list:

- Allocate memory for the new node.
- Determine the insertion point (you need to know only the new node's predecessor (`pPre`)
- Point the new node to its successor.
- Point the predecessor to the new node.

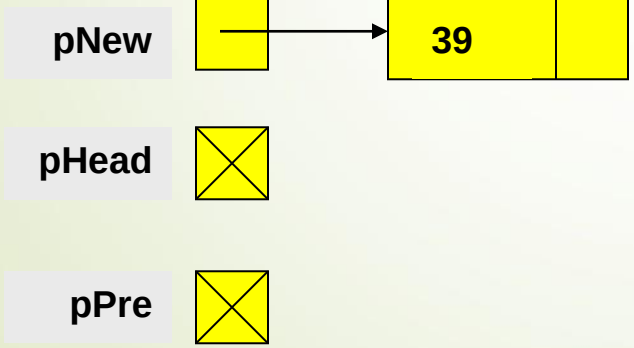
Pointer to the predecessor (`pPre`) can be in one of two states:

- it can contain the address of a node (i.e. you are adding somewhere after the first node – in the middle or at the end)
- it can be NULL (i.e. you are adding either to an empty list or at the beginning of the list)

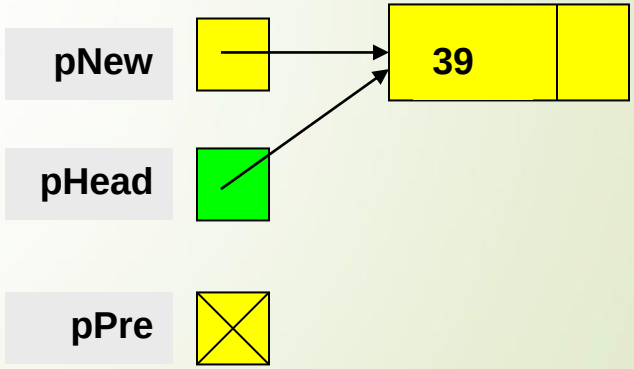
Adding Nodes to an Empty Linked List


```
pNew -> next = pHead;  // set link to NULL
pHead = pNew;// point list to first node
```

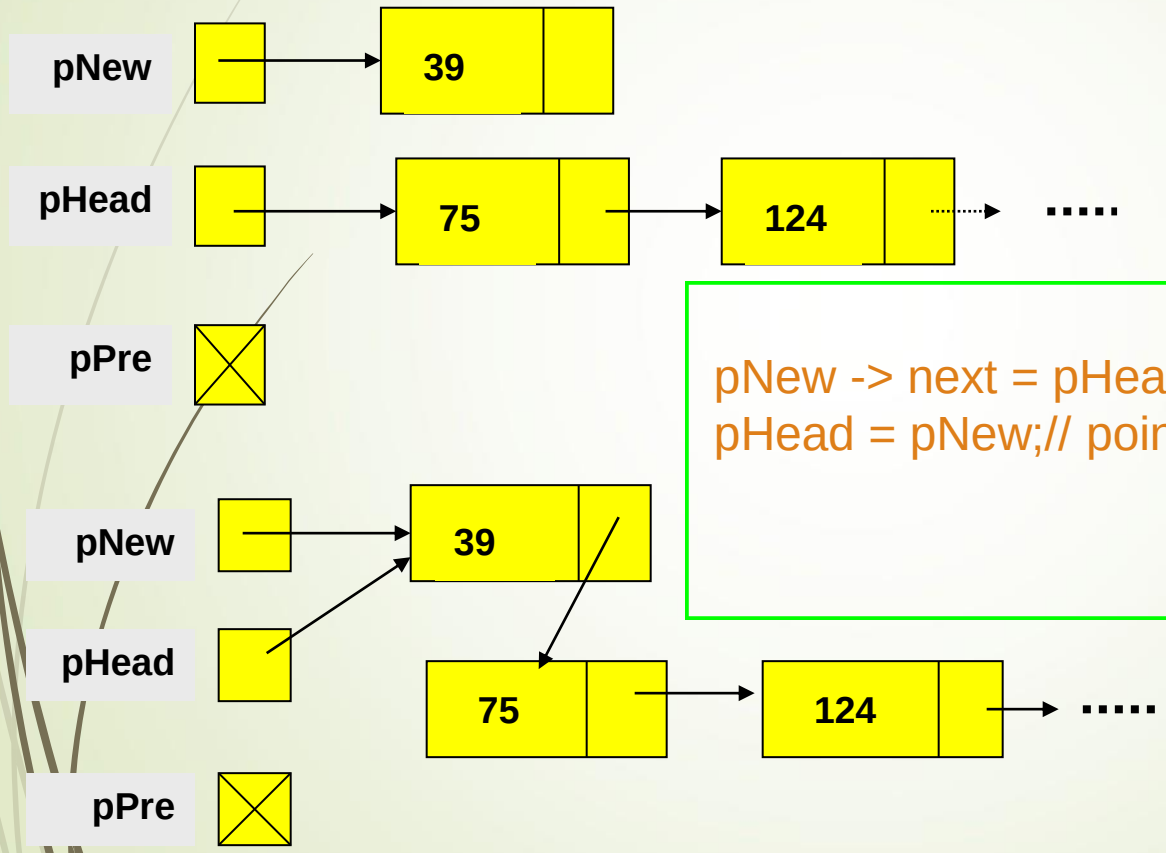
Before:



After:



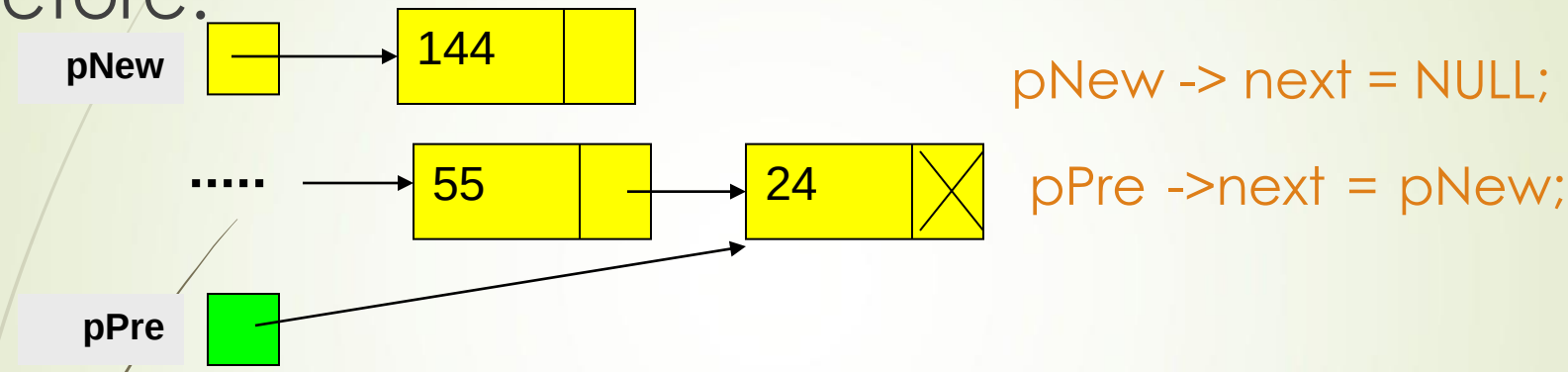
Adding a Node to the Beginning of a Linked List




```
pNew -> next = pHead;
pHead = pNew;// point list to first node
```

Adding a Node to the End of a Linked List

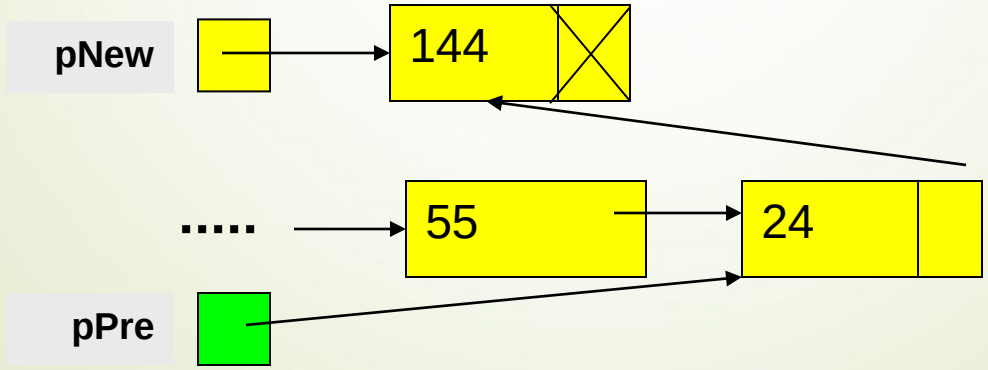
Before:



pNew -> next = NULL;

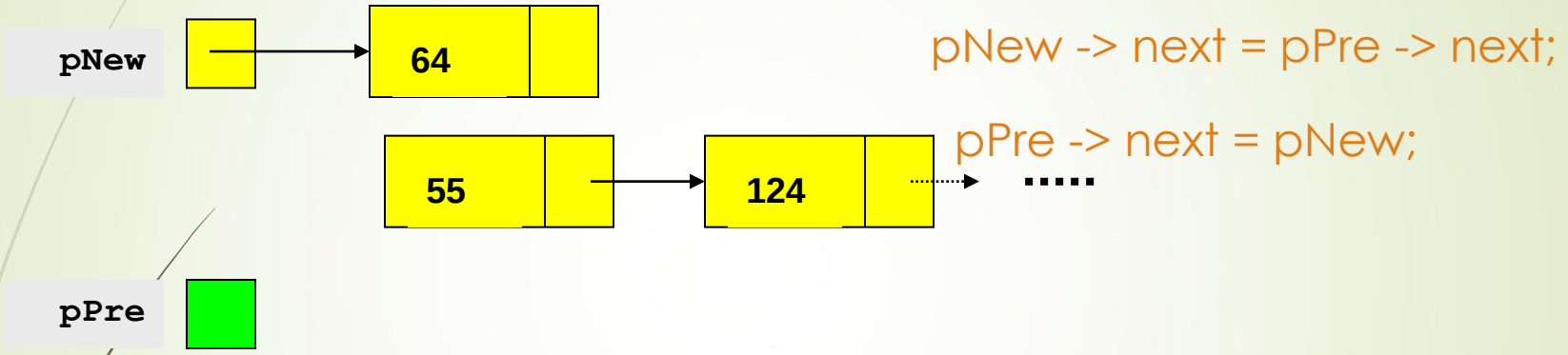
pPre ->next = pNew;

After:



Adding a Node to the Middle of a Linked List

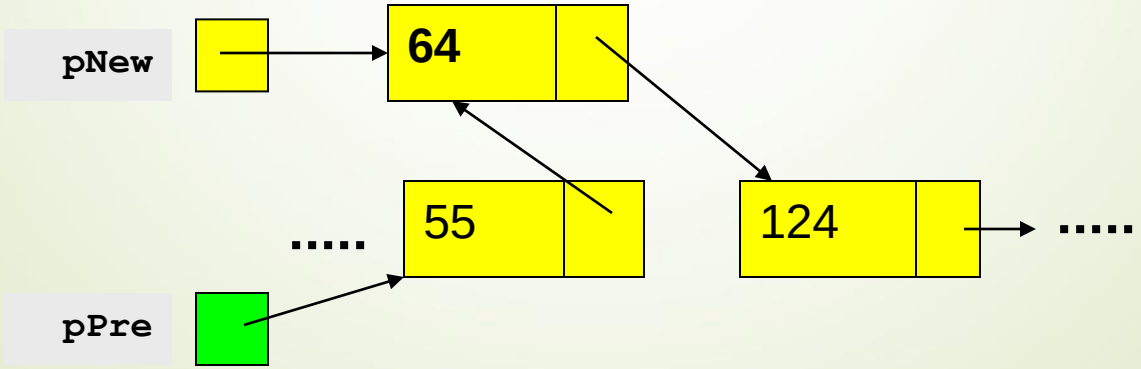
Before:



pNew -> next = pPre -> next;

pPre -> next = pNew;

After:



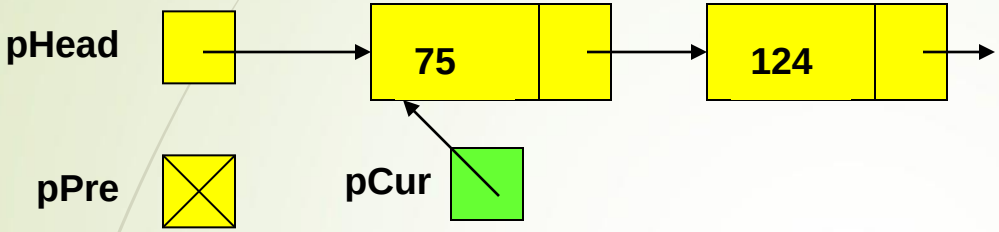
Deleting a Node from a Linked List

- Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
- Any node in the list can be deleted. Note that if the only node in the list is to be deleted, an empty list will result. In this case the head pointer will be set to NULL.

- To logically delete a node:
  - First locate the node itself (pCur) and its logical predecessor (pPre).
  - Change the predecessor's link field to point to the deleted node's successor (located at pCur -> next).
  - Recycle the node

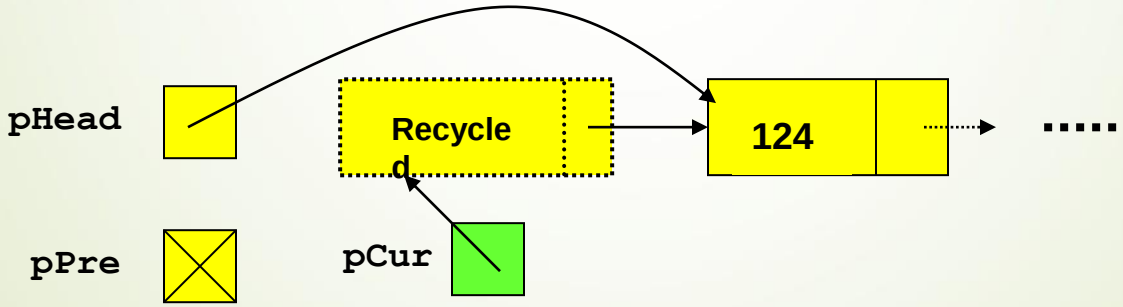
Deleting the First Node from a Linked List

Before:



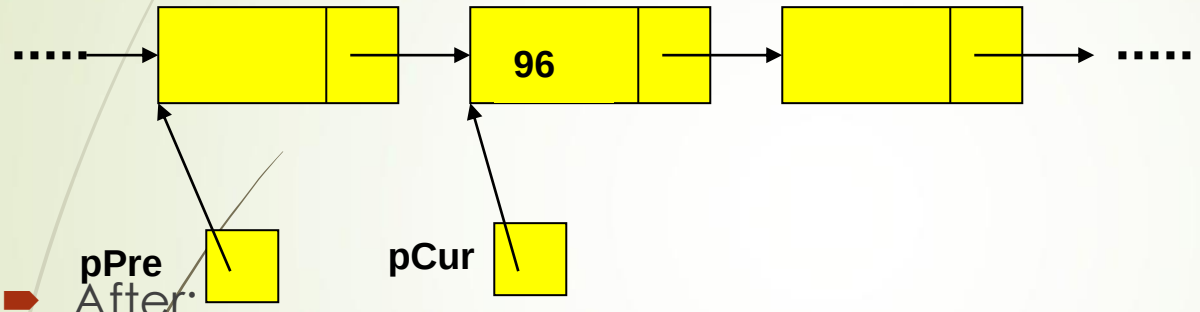

```
pHead = pCur -> next;
delete(pCur);
```

After:

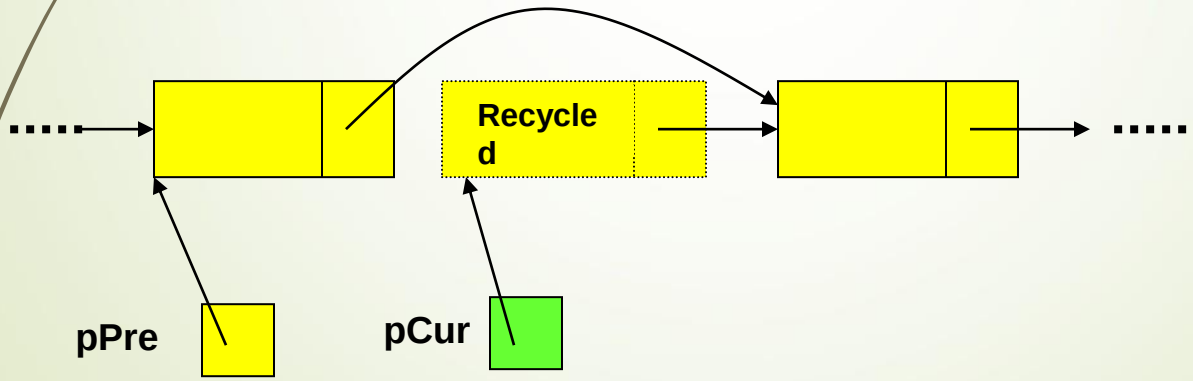


Deleting a Node from a Linked List – General Case

- Before:



- After:

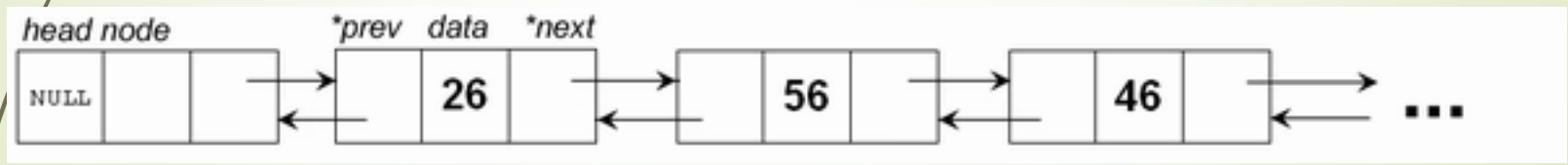


Code:

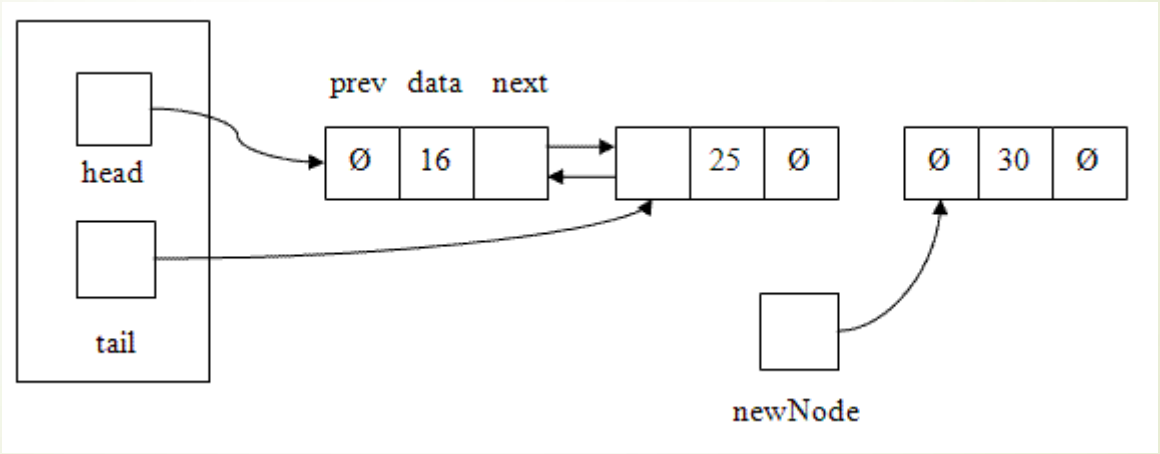
```
pPre -> next = pCur -> next;
delete(pCur);
```

Doubly Link List

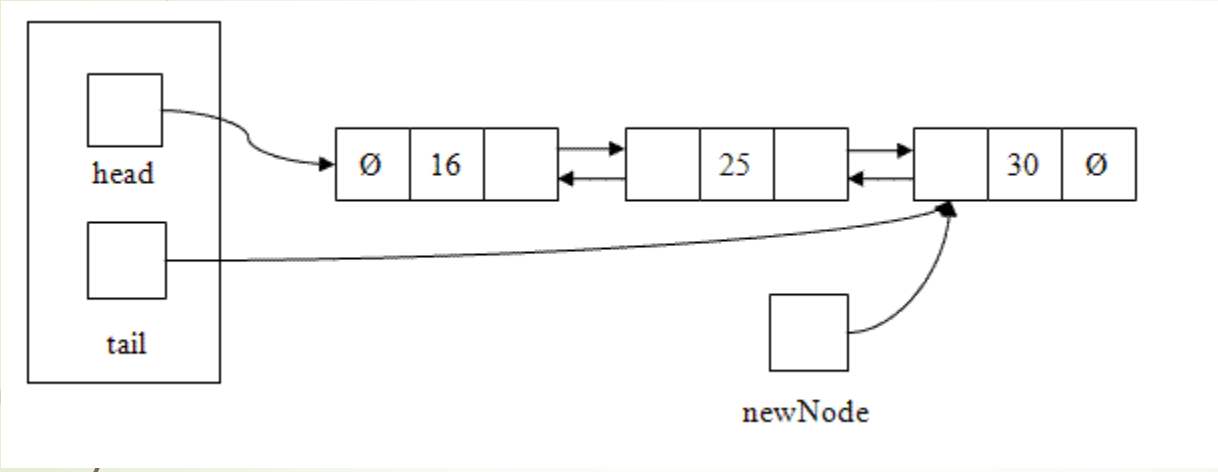
- It contains the data value
  it contains two pointer
- Next, Prev



Insert at Rear End (Last)



Insert at Rear



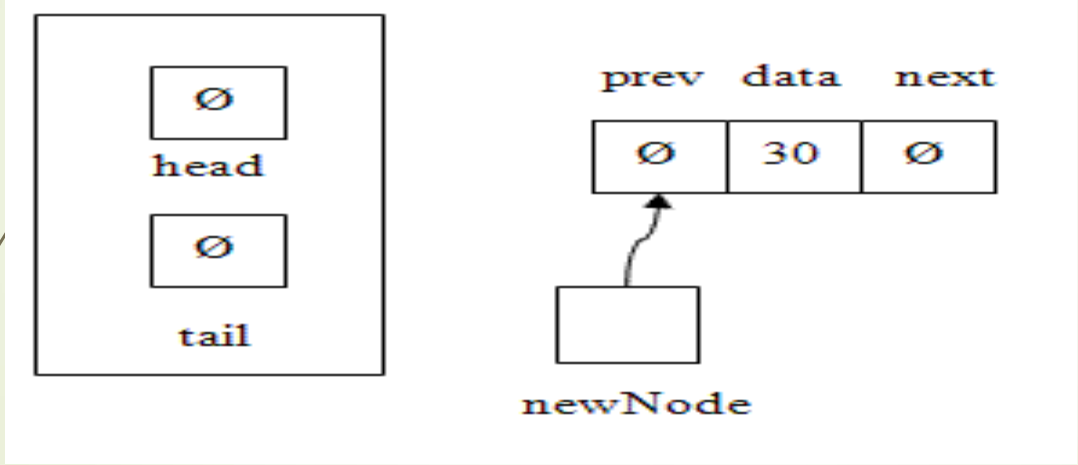
Insert at rear

newnode->prev= tail

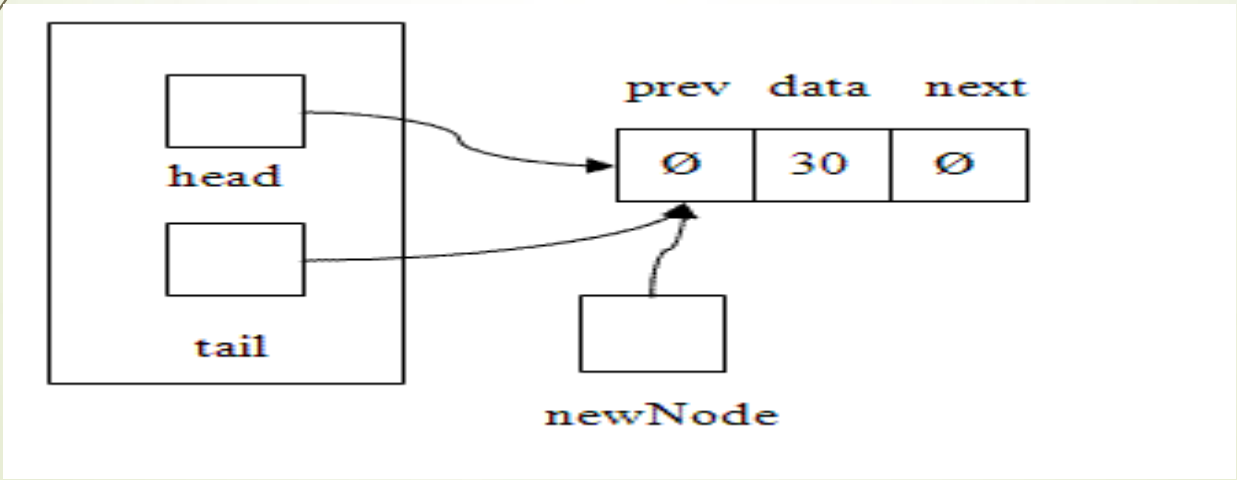
tail->next=newnode

tail=Newnode

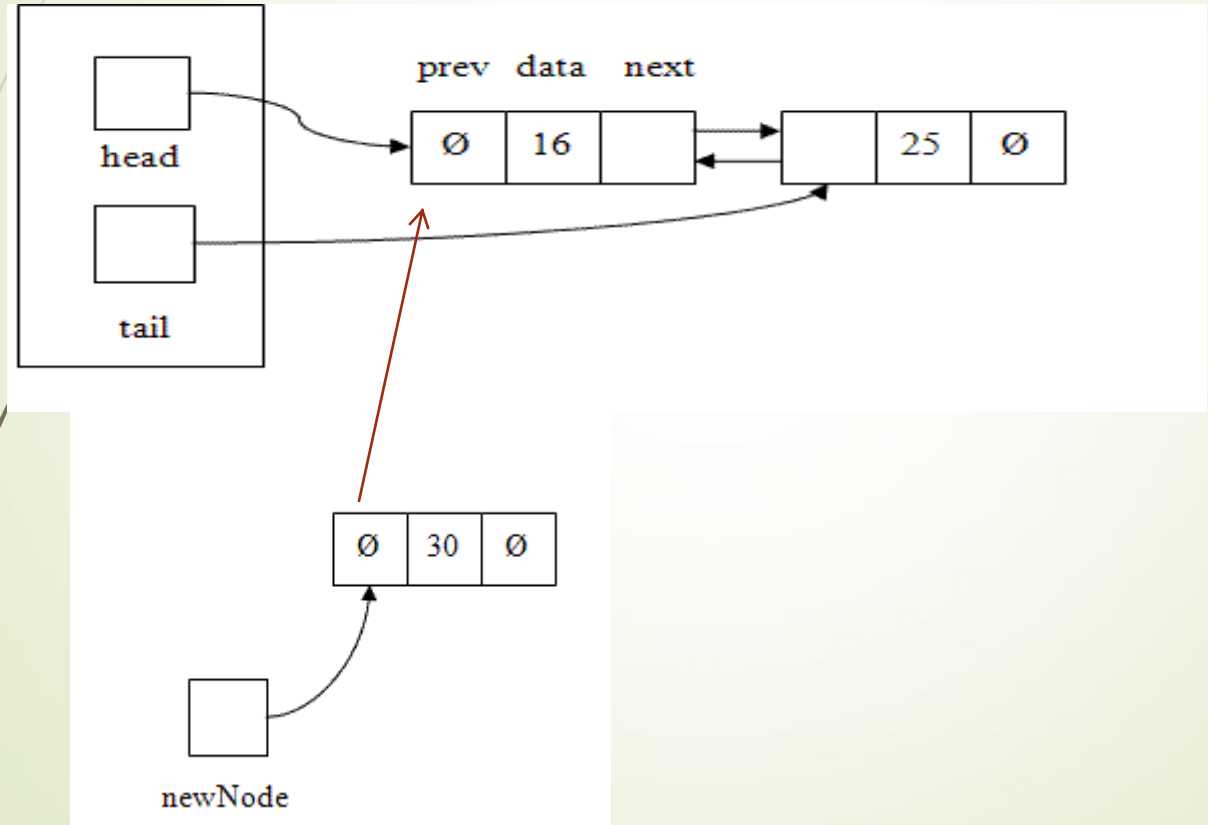
List is Empty



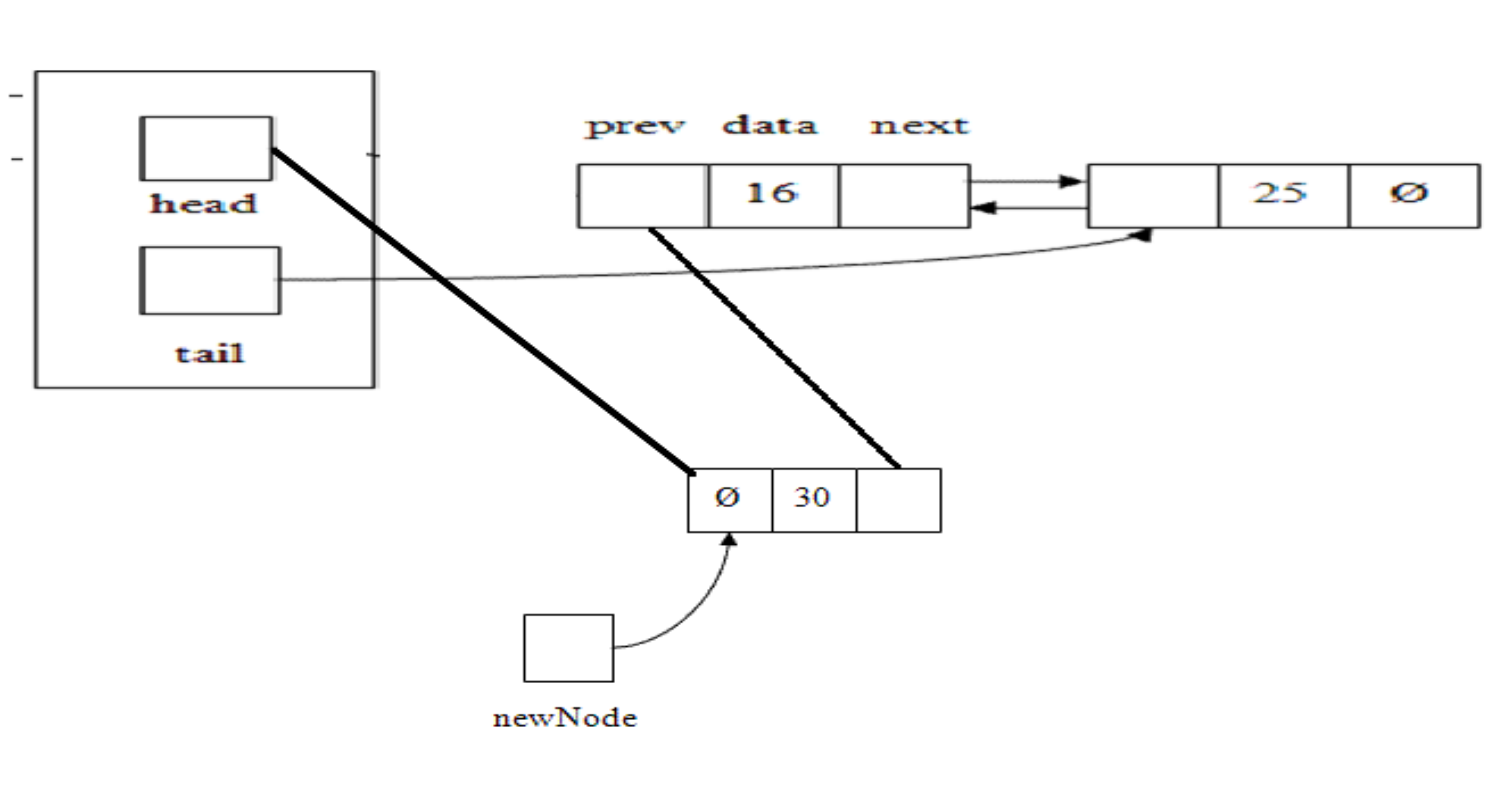
Newnode->prev=tail //(if tail=null)
head=newnode
tail=newnode
Newnode->next=null



Insert at First



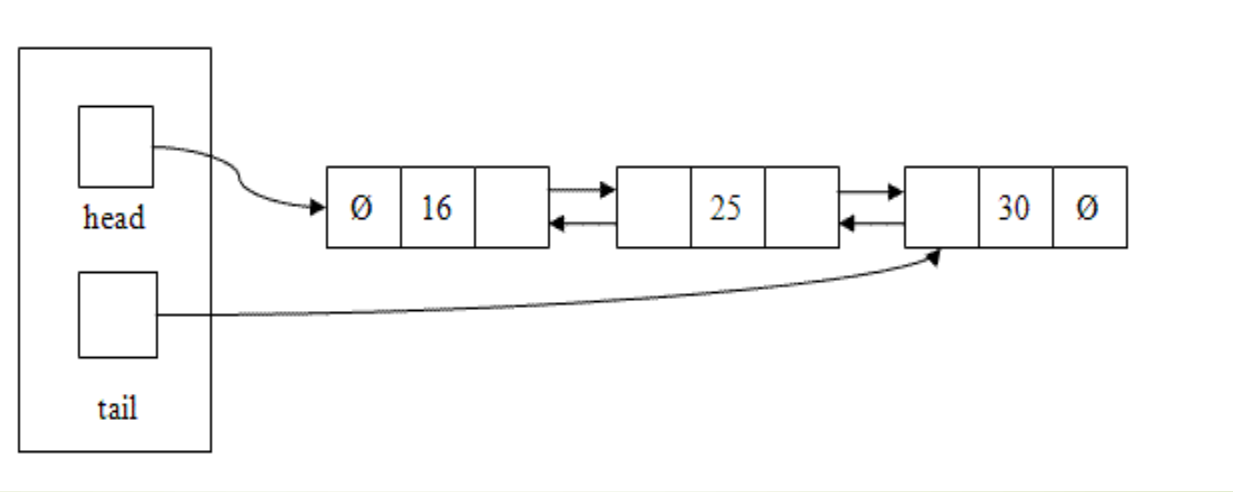
Newnode->next=head
head->prev=newnode
head=newnode



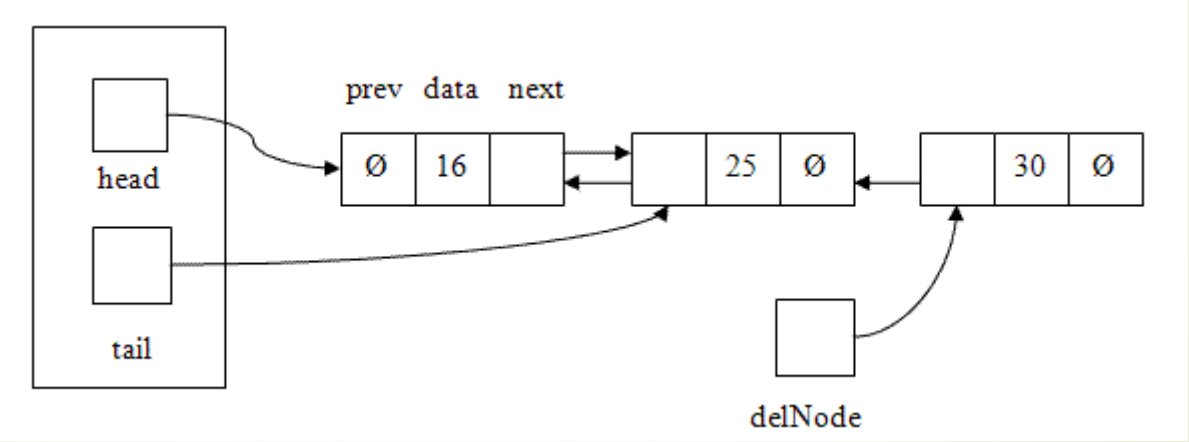
Insert at middle (some point)

- newnode->prev=current
- newnode->next=current->next
- current->next->prev=newnode
- current->next=newnode

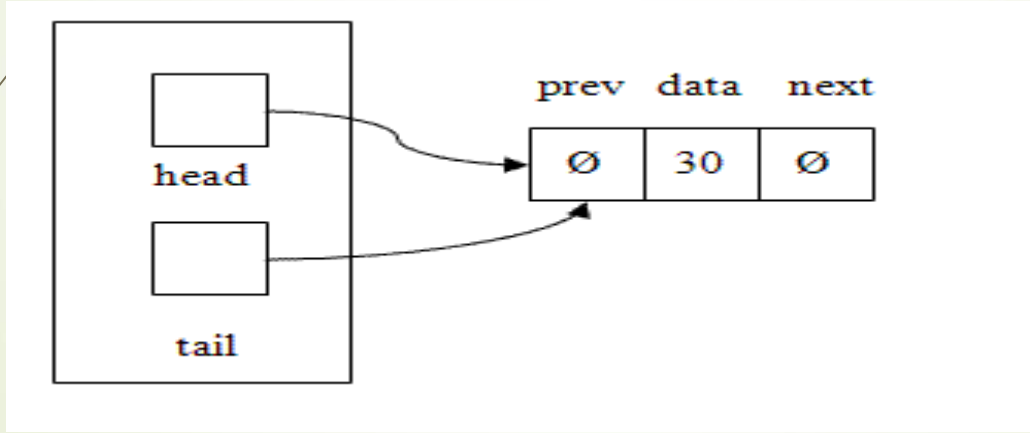
Delete at rear end (last)



lNode* delNode=tail
tail=delNode->prev
tail->next=null
delete delnode



Delete only One node



- lNode* delNode=tail
- tail=delNode->prev
- head=null // (tail==null)
- delete delNode

Delete from Front

- lNode* delNode=head
- head= delnode->next
- head->prev=null
- delete delnode

Delete From Middle (Any)

- curr->prev->next=curr->next
- curr->next-prev=curr->prev
- delete curr

Reading Material CS

- Schaum's Outlines: Chapter # 5

- D. S. Malik: Chapter # 5
- Mark A. Weiss: Chapter # 3
- Chapter 6, ADT, Data Structure and Problem solving with C++ by Larry Nyhoff.
- Chapter 5, Nell Dale 3rd Edition
- Chapter 8, C++ an introduction to Data Structures by Larry Nyhoff